Untersuchung von Strategien für verteiltes Last– und Ressourcenmanagement

Niels Reimer

reimer@informatik.tu-muenchen.de

16. Juni 1997

Zusammenfassung

Im Rahmen des MoDiS–Projektes, das sich mit der Konstruktion verteilter Systeme insbesondere verteilter Betriebssysteme beschäftigt, wurden zwei Experimentalsysteme (EVA und AdaM) entwickelt. Diese Systementwicklungsarbeiten sollten zum Erfahrungsgewinn und als Ausgangsbasis zukünftiger Systeme dienen. In beiden Systemen wurden spezielle Aspekte des Last– und Ressourcenmanagements auf jeweils unterschiedlichen Plattformen untersucht. Dieser technische Bericht stellt die beiden abgeschlossenen Arbeiten dar. Die Zielsetzung bei EVA war die Untersuchung von Strategien zur Lastplazierung in einem verteilten System. In AdaM wurden hingegen Ansätze zur verteilten Speicherverwaltung betrachtet. Ferner wurden Verfahren zur Nutzung von Alternativen der Realisierung von Datenobjekten untersucht, wobei die Auswahlentscheidung zur Laufzeit auf der Basis statisch und dynamisch ermittelter Information beruht. Das Hauptaugenmerk in diesem Bericht liegt allerdings auf den Randbedingungen des verteilten Systems unter denen die dafür verwendeten Strategien eingesetzt wurden und einer diesbezüglichen Bewertung der Strategien. Darüber hinaus werden die gewonnenen Erkenntnisse miteinander verknüpft und die Schlußfolgerungen für zukünftige Arbeiten präzisiert.

Inhaltsverzeichnis

1 Einleitung

Im Rahmen der Vorarbeiten und dann zum Teil fortgesetzt in den Arbeiten des Teilprojekts A8 im SFB 342 wurden die zwei Experimentalsysteme EVA[1] [1] und AdaM[2] [2] entwickelt. Beide Systeme untersuchen jeweils einen wesentlichen Teilaspekt im Bereich des Ressourcen– und Lastmanagements verteilter Rechensysteme. Ziel der Entwicklung dieser Systeme ist, eine verteilte Ausführungsumgebung für die Sprache INSEL [3, 4] zur Verfügung zu stellen. INSEL ist eine Ada–ähnliche, prozedurale, objektbasierte Sprache, in der insbesondere die dynamische Erzeugung und Auflösung von Komponenten konzeptionell verankert ist.

Das Hauptanliegen dieses Berichtes ist die Bewertung der verwendeten Strategien und Methoden hinsichtlich ihrer Verwend– und Übertragbarkeit für das im Teilprojekt A8 in Entwicklung befindliche, systemintegrierte Ressourcenmanagement. Dabei wird zunehmend die Koordination mehrerer verschiedener Ressourcen, umfassende Programmanalysen, die Anpassung der Strategien zur Laufzeit, sowie die Ausnutzung von Wechselwirkungen zwischen den entsprechenden Managementmaßnahmen im Brennpunkt des Interesses stehen.

EVA befaßt sich vor allem mit dem Problem der Lastplazierung; bei AdaM hingegen liegt das Hauptaugenmerk auf der Speicherverwaltung und der Nutzung von Alternativen zur Realisierung von Datenobjekten. Die wesentlichen Ziele, Lösungen und Verfahren in beiden Arbeiten sind in den nächsten Abschnitten dargestellt. Dabei ist die Sichtweise stark auf die verwendeten Strategien und die damit verwandten Aspekte ausgerichtet, um eine Bewertung diesbezüglich zu erreichen.

2 EVA

EVA ist ein Experimentalsystem, das Ausgangsbasis für die Entwicklung verteilter, paralleler Systeme gemäß der INSEL–Konzepte sein sollte. Als Realisierungsplattform wurde HP–UX auf einem HP–Cluster verwendet. Dabei stehen bei diesen Arbeiten die Aspekte zur Erzeugung und Plazierung leichtgewichtiger, feingranularer Aktivitätsträger (INSEL–Akteure) mit Hilfe eines systemintegrierten Lastmanagements im Vordergrund. Die Realisierung der Aktivitätsträger erfolgt als Threads, die seitens eines eigens entwickelten, verteilten Thread–Pakets, dem Distributed Thread Kernel (DTK) angeboten werden. Die Lastverteilung im DTK beschränkt sich auf die Plazierung der Threads bei ihrer Erzeugung, weshalb man genauer von einer Lastplazierung sprechen muß.

Ein System, das den DTK nutzt, besteht aus einer Menge kooperierender DTK–Tasks (Prozesse), die die DTK–Threads beinhalten und verwalten. Dabei wird genau eine DTK–Task auf jedem der beteiligten Rechnerknoten plaziert. Da das Ressourcenmanagement (Lastplazierung) nicht anwendungsintegriert, das heißt nicht durch die erzeugten Komponenten des INSEL–Systems, durchgeführt werden soll, wurde es in die DTK–Tasks integriert.

[1] Experimentalsystem für verteilte Anwendungen

[2] Adaptives Ressourcenmanagement unter Mach3.0

Jedesmal wenn das in Ausführung befindliche Programm bzw. ein Thread, der in einer DTK–Task läuft, einen weiteren Thread erzeugen will, meldet er sich bei seiner ihm zugeordneten DTK–Task. Diese bestimmt dann unter Verwendung einer Strategie eine geeignete Ziel–DTK–Task, auf der dann die Aktivität plaziert und ausgeführt wird. Dadurch wird eine transparente Nutzung der Hardware erreicht und somit der Zielsetzung einer ortstransparenten Realisierung entsprochen.

Die im DTK integrierten Strategien werden detailliert im nächsten Abschnitt beschrieben, für darüber hinausgehende, vertiefende Beschreibungen sei auf [1] verwiesen.

2.1 Strategien in EVA/DTK

Im DTK sind 5 Strategien implementiert, die alle in ihrem Entscheidungsprozeß die relative Leistungsfähigkeit der Knoten berücksichtigen. Dies geschieht mit Hilfe der Leistungsmeßzahl l_i, die beim Start der DTK–Task i bestimmt wird. Beim Start einer DTK–Task werden die Meßzahlen verschickt und als Antwort erhält die neu am System teilnehmende DTK–Task die Meßzahlen der anderen am DTK–System beteiligten Tasks. Da von einer exklusiven Nutzung der jeweiligen Knoten für die DTK–Task ausgegangen wird, entfällt eine Kalibrierung zu einem späteren Zeitpunkt. Die Leistungsmeßzahlen können daher als Konstanten angesehen werden.

Die Strategien im einzelnen sind: *Random*, ein zufallsgesteuertes Plazierungsverfahren; *OptimalLocal*, ein Verfahren, das nur die Last betrachtet, die vom lokalen Knoten auf anderen Knoten plaziert wurde; *Load*, ein Verfahren mit globaler Systemsicht der Lastverhältnisse; *RBidding*[3], ein Empfänger–initiiertes Verfahren zur Lastverteilung und *SBidding*[4], ein Sender–initiiertes Auktionsverfahren. Wobei hervorzuheben ist, daß die Bidding–Strategien die Thread–Erzeugung verzögern können, sofern alle DTK–Tasks ausgelastet sind und daher eine weitere Erzeugung das Gesamtsystem nur verlangsamen würde.

Random

Der Entscheidungsprozeß für die Plazierung eines zu erzeugenden Threads in einer DTK–Task ist vergleichsweise einfach. Es wird die Summe der Leistungsmeßzahlen $s = \sum_{i=1}^{n} l_i$ gebildet und eine Zufallszahl x im Intervall von Null bis zur Summe s erzeugt. Der Thread wird dann auf der DTK–Task plaziert, mit deren Beitrag zur Summe die sich ergebende Teilsumme gerade die ermittelte Zufallszahl erreicht oder überbietet. Das heißt: $Index\ des\ Plazierungsziels = \min\{k \mid k \in [1:n] \wedge Zufallszahl\ x \in [0;s] \ \wedge \ x \leq \sum_{i=1}^{k} l_i\}$ Diese einfache Strategie kommt ohne Kenntnisse der Lastzustände des Systems aus und läßt sich davon auch nicht beeinflussen. Daher wird sie oft als Referenzstrategie herangezogen.

[3]Receiver–initiated Bidding

[4]Sender–initiated Bidding

OptimalLocal

Bei der Strategie OptimalLocal wird im Gegensatz zu Random zumindest eine einseitige Sicht der Systemlastverhältnisse betrachtet. Dazu führt jede DTK–Task zusätzlich zu den Leistungsmeßzahlen einen Zähler pro beteiligter DTK–Task, der die Anzahl der auf dieser DTK–Task erzeugten und laufenden DTK–Threads angibt. Wenn ein neuer DTK–Thread erzeugt bzw. ein Plazierungsziel ermittelt werden soll, bestimmt die Strategie die Quotienten aus den jeweiligen Zählern und den entsprechenden Leistungsmeßzahlen. Es wird dann die DTK–Task mit dem kleinsten Quotienten ausgewählt, die Thread–Erzeugung dort veranlaßt und der entsprechende Zähler lokal erhöht. Wenn ein Thread beendet wird, erhält die die Erzeugung veranlassende DTK–Task eine Rückmeldung und erniedrigt den lokalen Zähler wieder.

Die Strategie versucht also, die auf ihrem Knoten erzeugten Lastverursacher möglichst gleichmäßig (gemäß der relativen Leistungsfähigkeit) auf das System zu verteilen. Dabei wird bedingt durch die lokale Systemsicht die Last, die sich zwei andere Tasks gegenseitig aufbürden, ignoriert. Ferner wird davon ausgegangen, daß jede Thread–Erzeugung und –Bearbeitung die gleiche Belastung mit sich bringt. Nur unter dieser Voraussetzung kann die Strategie OptimalLocal im Allgemeinen eine gute Lastverteilung erreichen. Zudem werden so regelmäßige Kommunikationen zwischen den DTK–Tasks vermieden, da nur bei Beendigung eines Threads eine Nachricht verschickt werden muß.

Load

Die Strategie Load bemüht sich um eine globale Systemsicht. Jede DTK–Task berechnet ihren lokalen Lastindex (Anzahl der laufenden Threads) in regelmäßigen Zeitabständen und meldet diesen allen beteiligten DTK–Tasks, wenn er sich, gemessen am zuletzt mitgeteilten Index, über gewisse Beträge hinaus geändert hat.

Jedesmal wenn ein Thread erzeugt werden soll, wird für alle DTK–Tasks der Quotient zwischen Lastindex und Leistungsmeßzahl gebildet und die DTK–Task mit dem Minimum ausgewählt. Um die Systemsicht möglichst aktuell zu halten und ein Überfluten von nur gering ausgelasteten DTK–Tasks zu vermeiden, wird anschließend noch lokal in der versendenden DTK–Task der Lastindex des gewählten Ziels erhöht.

Bedingt durch die Wahl der Größe des Zeitintervalls, in dem der eigene Lastindex bestimmt wird, kann das Nachrichtenaufkommen bei hoher Dynamik des Systems beträchtlich werden. Allerdings leidet bei zu großer Wahl des Intervalls die Aktualität der Systemsicht.

RBidding

Während die ersten drei Strategien recht einfach sind, gestalten sich die Bidding–Strategien wesentlich komplizierter. Die implementierte RBidding– und SBidding–Strategie stellt jeweils eine Modifikation der allgemeinen Auktionsstrategie dar, die in [5] beschrieben ist. Der Hauptunterschied liegt im wesentlichen in der Eigenschaft, die Thread–Erzeugung verzögern zu können, um so bei einem ausgelasteten System keine Überlast aufkommen zu lassen und den Mangel an Migrationsmechanismen für DTK–Threads durch verzögerte Plazierung auf später frei werdende DTK–Tasks zu lindern. Um die eigene Auslastungssituation zu ermitteln, bestimmt jede DTK–Task periodisch ihren Lastindex (Anzahl der laufenden Threads) und kennt einen Schwellwert bezüglich dieses Indexes. Liegt der Index unterhalb dieses Wertes, so gilt sie als unterlastet; liegt er oberhalb, so gilt sie als überlastet. Die Grundidee beim RBidding ist, daß unterlastete DTK–Tasks Anfragen nach Arbeit stellen und dann von überlasteten DTK–Tasks Thread–Erzeugungen übernehmen und diese somit entlasten. Dabei erfolgt das Angebot zur Nutzung der Kapazität nur an soviele DTK–Tasks, daß im Falle des Empfangs eines Thread–Erzeugungsauftrages pro ausgegebenem Angebot, die anbietende DTK–Task gerade nicht mehr unterlastet wäre. Nach Ablauf einer gewissen Zeitspanne werden nicht wahrgenommene Angebote zurückgezogen und die freie Kapazität anderen DTK–Tasks angeboten. Auf die Art wird die Liste der beteiligten DTK–Tasks sukzessive zyklisch durchwandert.

Überlastete DTK–Tasks vergeben Thread–Erzeugungen an anfragende DTK–Tasks ab oder, falls keine solche verfügbar ist, verzögern sie ihre Thread–Erzeugungen, indem sie die Aufträge in eine Warteschlange einreihen. Wenn sich dann eine unterlastete DTK–Task mit einem Kapazitätsangebot meldet, wird ein Erzeugungsauftrag aus der Warteschlange gestrichen und an die DTK–Task abgegeben. Falls die Warteschlange leer ist, wird nur die Anfrage vermerkt. Für nähere Implementierungsdetails und einstellbare Parameter sei an dieser Stelle auf [1] verwiesen.

SBidding

Der Unterschied beim SBidding gegenüber dem RBidding liegt darin, daß überlastete DTK–Tasks diese Tatsache durch eine systemweite Nachricht allen am System beteiligten DTK–Tasks mitteilen. Sie reihen dann ihre Thread–Erzeugungsaufträge vorerst in eine Warteschlange ein. Die Warteschlange arbeiten sie erst dann ab, wenn sich ihre Lastsituation entspannt oder Kapazitätsangebote als Antwort auf die Überlastungsnachricht eintreffen. Aus dem auf die Art erhaltenen systemweiten Kapazitätsangebot könnten nun noch selektiv die jeweils leistungsfähigsten DTK–Tasks ausgesucht werden. Messungen an einer prototypischen Implementierung zeigten aber, daß die Verzögerung durch ein Abwarten eingehender Angebote und ein Sortieren zur Auswahl der leistungsstärksten DTK–Tasks die Vorteile einer so optimierten Zuordnung bei weitem aufwiegen. So wird in der vorliegenden Implementierung unterstellt, daß die DTK–Task, die sich zuerst meldet, die Leistungsstärkste ist und damit die beste Wahl darstellt. Zudem wird so die Zeit bis zum Eingang weiterer Angebote genutzt, um Thread–Erzeugungsaufträge zu verteilen. Auch hier sei für die Erläuterung und Vertiefung weiterführender Implementationsdetails auf [1] hingewiesen.

2.2 Meßergebnisse zu den Lastplazierungsstrategien

Die Bewertung und der Vergleich der Leistungsfähigkeit der einzelnen Strategien erfolgte anhand von Benchmarks. Dabei wurden zwei typische Thread–Erzeugungsmuster verwendet:

- **Flat:** Eine flache Thread–Hierarchie, bei der ein Thread alle anderen erzeugt und dann auf deren Rückkehr wartet.
- **BTree:** Eine balanzierte Binärbaum-artige Thread–Hierarchie, bei der jeder erzeugte Thread, sein ihm zugewiesenes Arbeitspaket abarbeitet und dann die Menge der von ihm (ggf. indirekt) noch zu erzeugenden Threads möglichst gleichmäßig auf zwei Mengen verteilt. Dann erzeugt er zwei Threads, denen er jeweils eine Menge übergibt. Anschließend wartet er auf deren Rückkehr.

In den Benchmarks wurde jeweils ein konstant gehaltenes Gesamtarbeitsvolumen geleistet, so daß durch die Wahl der Granularität (Größe des jeweiligen Arbeitspaketes) die Anzahl der Aktivitätsträger dazu umgekehrt proportional ist. Die Strategien werden mit einem weitem Spektrum bezüglich des Parallelitätsgrades konfrontiert, das von vielen kleinen Aktivitätsträgern (16384 Threads mit je einem Paket) bis hin zu wenigen längerlaufenden Aktivitätsträgern (128 Threads mit je 128 Paketen) reicht. Zusätzlich wurden Messungen durchgeführt, bei denen das Lastaufkommen je Thread zwischen 1 und 16 Paketen schwankte. Dabei ist zu beachten, daß eine der Annahmen der Strategie OptimalLocal nicht erfüllt ist. Die Ergebnisse, die mit den beiden Thread–Erzeugungsmustern gewonnen wurden, liegen bezüglich der Güte der Auslastung etwa in der gleichen Größenordnung. OptimalLocal und RBidding erzielen bei der Thread–Erzeugung gemäß Flat die besten Resultate, wobei OptimalLocal umso schlechtere Resultate erzeugt, je stärker die Paketanzahl je Thread variiert. Zudem zeigt die Strategie Random stets wesentlich schlechtere Lastverteilungen als ihre Konkurrenten, was auch den Erwartungen entspricht, da keinerlei Lastinformation verarbeitet wird. Weitere Beschreibungen und Ergebnisse der Benchmark–Szenarien sind in [1] dokumentiert. Exemplarisch für die durchgeführten Messungen sind in Abbildung 1 hier nur die Ergebnisse von BTree mit jeweils einheitlichem Lastaufkommen je Thread dargestellt.

Um ein Maß für die Auslastung in einem heterogenen System angeben zu können, führen wir hier den Begriff der relativen Auslastung ein. Die relative Auslastung wird wie folgt berechnet: Als obere Schranke bei der Abschätzung der Leistungsfähigkeit des benutzten Rechnerverbundes (19 unterschiedliche HP–Arbeitsstationen der Serie HP 9000/7xx) wird die Summe der Leistung der einzelnen Arbeitsstationen angesetzt. Die Einzelbeiträge wurden ermittelt, indem das Benchmark–Programm ausschließlich mit einer einzelnen DTK–Task auf dem jeweiligen Rechnertyp ausgeführt und die Laufzeit gemessen wurde. Die relative Auslastung ergibt sich dann jeweils aus dem Quotienten der oberen Leistungsschranke und der erbrachten Leistung (16384 Pakete geteilt durch die benötigte Laufzeit des Benchmarks–Programms auf dem gesamten Verbund).

Man sieht deutlich, daß die lastsensitiven Strategien erst akzeptable Ergebnisse liefern, wenn die Last pro Knoten ein gewisses Maß erreicht hat. Das heißt, erst bei

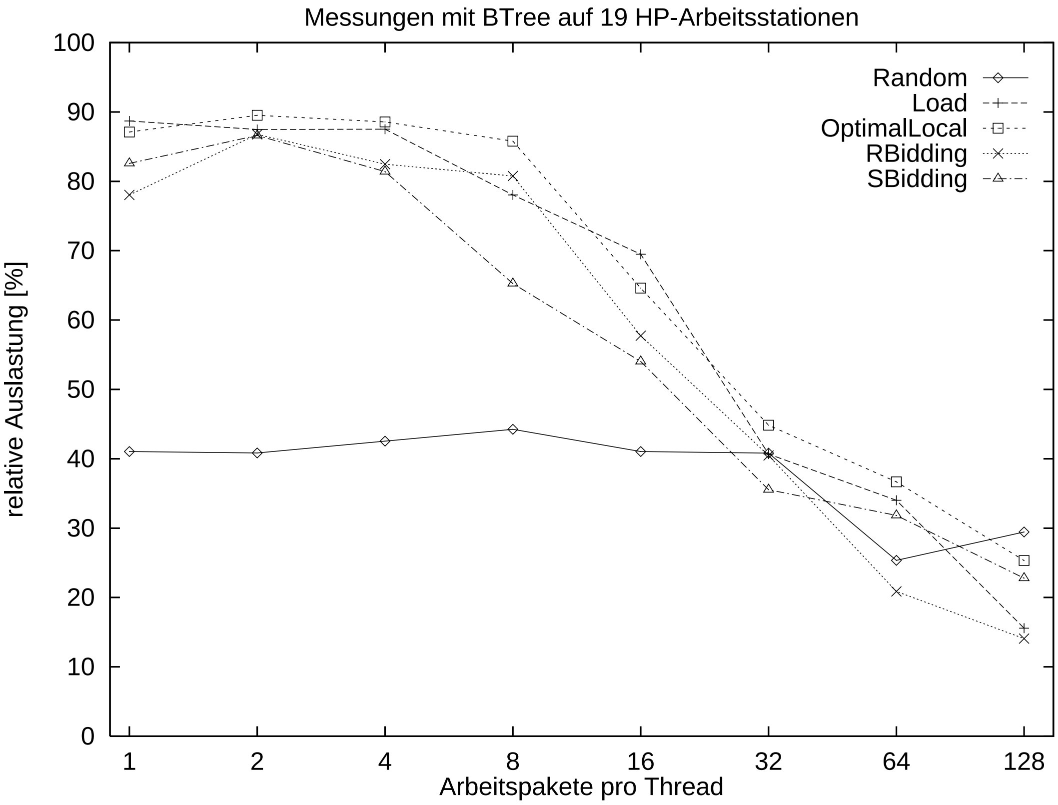

Abbildung 1: Erreichte relative Auslastung des Rechnerverbundes in Abhängigkeit der Granularität der Lasterzeuger (Pakete pro Thread) bei Verwendung verschiedener Strategien

mehr als ca. 100 Threads pro Rechnerknoten[5] läßt sich eine relative Auslastung über 75 % erzielen. Benchmarks mit weniger Rechnerknoten zeigen, wegen der höheren Thread–Anzahl pro Knoten, vergleichsweise bessere Auslastungswerte bei gleich großen Arbeitspaketen. Bei einer geringen Anzahl längerlaufender Aktivitätsträger pro DTK–Task hat die Lastplazierung unter Umständen deutliche Probleme, eine gleichmäßige Auslastung zu erreichen, die die positiven Effekte der Verringerung des Managementaufwandes dominieren. Die auf die Auslastung bezogenen nachteiligen Effekte einer groben Granularität zeigen sich auch bei der Strategie Random, jedoch ist diese Strategie wesentlich robuster bzw. unempfindlicher dagegen.

2.3 Bewertung der Strategien in EVA

Insgesamt konnten durch diese Meßreihen mit einem simulierten parallelen Anwendungsverhalten gezeigt werden, daß die Strategien zur integrierten Lastverwaltung eine sinnvolle Ausgangsbasis darstellen, da trotz der Beschränkung auf Aktivitätsplazierungen 90 % der maximalen Leistungsschranke erreicht wurden. Allerdings

[5] 16384 Pakete gebündelt zu 8 Paketen pro Thread ergibt 2048 Threads, diese verteilt auf 19 Rechnerknoten ergeben im Mittel 107 Threads pro Knoten.

muß auch daraufhingewiesen werden, daß die Aktivitätsträger im Rahmen der Vergleichsmessungen weder eine weitere Kommunikation außer der Parameterübergabe zu Beginn und der Synchronisation am Ende ihrer Tätigkeit durchführen, noch verteilte Datenobjekte gemeinsam nutzen und darüber hinaus eine sehr regelmäßige und statische Abhängigkeitsstruktur zeigen. Die Strategien ignorieren solche Eigenschaften und Abhängigkeiten zwischen den zu verwaltenden Lastkomponenten und betrachten einseitig nur die Ressourcenkapazität der Hardware, an die sie durch die Integration in den DTK fest gebunden sind. Zudem sind die gemachten Einschränkungen und Annahmen beim Entwurf der Meßreihen für einen prinzipiellen Nachweis der Angemessenheit der Lastplazierungsstrategien hinsichtlich der Verteilung von Last gedacht, jedoch für ein verteiltes System unter realen Bedingungen sind sie unrealistisch und nicht adäquat. Daher müssen für eine Weiterentwicklung der Strategien zusätzlich zur Sichtweise auf das Ressourcenangebot auch die Information der Abhängigkeiten der Ressourcennachfrage d. h. der zu plazierenden und auch zu migrierenden Objekte genutzt und miteinbezogen werden. Dadurch erscheint eine derartige ausschließliche Bindung des Ressourcenmanagements an die Hardware–Stellen als ungeeignet. Die Integration des Lastmanagements, sowohl in die Stellenkerne (Ressourcenanbieter) als auch in Manager, die den Aktivitätsträgern (Ressourcennutzer) assoziert sind, erscheint als der einzuschlagende Weg zu effizienteren Systemen.

3 AdaM

Das Experimentalsystem AdaM [2] baut auf dem Mikrokern von Mach3.0 auf und enthält einen INSEL–Übersetzer sowie ein verteiltes Laufzeitsystem, das im wesentlichen eine Komponenten– und eine Speicherverwaltung bietet. Die Aktivitätsträger der INSEL–Programme werden ebenfalls über Threads realisiert. Die Komponentenverwaltung plaziert diese über ein primitives zufallsgesteuertes Verfahren und löst sie bei Beendigung ihrer Tätigkeit auf. Die Speicherverwaltung stellt allen übersetzten Komponenten mit Hilfe verteilter Distributed Shared Memory–Server, die als stellengebundene Wurzelverwalter realisiert sind, einen globalen Adreßraum zur Verfügung. Zudem wird jedem Aktivitätsträger ein Speicherverwalter zugeordnet, der über eine vom Wurzelverwalter zugeteilte Speichermenge verfügt. Dieser Speicherverwalter bedient alle Anforderungen und Freigaben seines Aktivitätsträgers. Kann er sie nicht erfüllen, fordert er von seinem Wurzelverwalter Speicher nach. Dieser kann im Mangelfall von den anderen, lokalen, aktivitätsgebundenen Verwaltern die Rückgabe ungenutzter Speicherblöcke anfordern. Sollte auch dies nicht genügen, so fordert er von anderen Wurzelverwaltern entfernte Speicherseiten zusätzlich an und teilt den so insgesamt erhaltenen Speicher dann neu zu. Auf die dafür nötigen Mechanismen und Strategien der Wurzelverwalter zur Speicherverwaltung soll aber hier nicht vertiefend eingegangen werden. Im Folgenden werden die strategieschen Aspekte der dynamischen, adaptiven Realisierung von Datenobjekten behandelt.

So wurde für die Realisierung von Datenobjekten ohne exklusive Lesezugriffe die Möglichkeit der dynamischen Replikation geschaffen, das heißt, diese Objekte werden gegebenenfalls zur Laufzeit repliziert. Die Replikate werden von der Stelle mit der initialen Realisierung des Datenobjektes mittels eines Protokolls zur Sicherung der schwachen Konsistenzeigenschaft auf aktuellem Stand gehalten. Dadurch können entfernte Nutzungsaufrufe durch lokale Zugriffe ersetzt werden. Zeigt sich im weiteren Verlauf der Nutzung, daß diese Realisierungsvariante unrentabel ist, so wird das Replikat wieder aufgelöst und der entfernte Zugriffmodus verwendet. Für die Entscheidung zur Replikation bzw. deren Auflösung wurden zwei Strategien, die auf Heuristiken basieren, entwickelt und untersucht. Für Monitor–Depots (Datenobjekte mit wechselseitigem Ausschluß der Zugriffsoperationen) wurde eine Strategie zur Migration entwickelt. Alle drei Strategien werden im folgenden Abschnitt näher beleuchtet.

3.1 Strategien in AdaM

Die eingesetzten Speicherverwaltungstechniken sollen insbesondere den schnellen (vorzugsweise lokalen) Zugriff auf die Repräsentationen der Datenkomponenten, die von den Aktivitätsträgern lokal erzeugt wurden, erlauben. Dazu wurden für Datenkomponenten, für die eine dynamische Erzeugung von Replikaten vorgesehen ist, zwei Strategien (*Server–* und *Client–Strategie*) entwickelt. Für Datenkomponenten, die dem Monitor–Konzept entsprechen, wurde die Möglichkeit der Migration geschaffen und damit auch die nötige *Migrationsstrategie* zur Bestimmung des Migrationszieles und –zeitpunktes. Die verwendeten Heuristiken und ihre einstellbaren Parameter wurden in umfangreichen Benchmarks untersucht, entwickelt und gegeneinander abgestimmt [2].

Server–Strategie

Die Server–Strategie ist an die Verwaltungskomponente der initialen Realisierung des Datenobjektes gebunden. Die Verwaltungskomponente bedient alle entfernten Zugriffe und gewinnt damit ein Bild über das Nutzungsverhalten. Die Server–Strategie entscheidet auf dieser Grundlage, ob das Datenobjekt auf andere Knoten des Systems von denen ein Zugriff erfolgt, repliziert werden soll. Ein Replikat erfordert bei ausgeführten Schreibzugriffen ein Propagierung der Änderungen, um die Konsistenz zu wahren. Daher kann eine Replikation nur rentabel sein, wenn Schreibzugriffe eher selten und der Umfang der Änderungen begrenzt ist. Ferner sollten Knoten die Replikate verwalten dadurch auch nicht überlastet werden. Somit wird nur repliziert, wenn alle drei folgende Bedingungen erfüllt sind:

1. Der Anteil der Schreiboperationen an der Gesamtanzahl der Zugriffsoperationen ist kleiner als das Verhältnis eines einstellbaren Parameters *ChangeRatio* zur Anzahl der Rechnerknoten mit einem Replikat des entsprechenden Datenobjekts.

2. Die durchschnittliche Größe einer Änderung am Datenobjekt ist kleiner als eine einstellbare Konstante $\mathit{MaxDiffSize}_{Server}$, die die maximale Änderung beschreibt, die bezüglich des Aufwandes der Aktualisierung der Replikate toleriert wird.

3. Der Zielknoten für die Replikation ist nicht überlastet.

Empirische Versuche zeigten für den Parameter $ChangeRatio$ ein Leistungsoptimum bei 40 %. Allerdings ist zu berücksichtigen, daß für den Wert von $ChangeRatio$ ebenso wie für den Parameter $MaxDiffSize_{Server}$ gilt, daß die optimalen Werte stark von der verwendeten Hardware–Umgebung abhängen.

Client–Strategie

Die Client–Strategie ist im Gegensatz zur Server–Strategie an das Replikat eines Datenobjekts gebunden und gestattet einen lokalen Zugriff. Ihr obliegt die Entscheidung, das Replikat zu nutzen oder es aufzulösen und wieder zum entfernten Zugriffsmodus zurückzukehren. Analog zur Server–Strategie müssen für die Nutzung des lokal vorliegenden Replikats ebenfalls drei Bedingungen erfüllt sein:

1. Die Anzahl der Nachrichten, die zur Konsistenzerhaltung des Replikats zwischen dem eigenen Knoten und dem Knoten mit der initialen Inkarnation des Datenobjektes ausgetauscht werden, muß kleiner oder gleich der doppelten Anzahl der lokalen Zugriffe sein.

2. Die durchschnittliche Größe einer Änderung am Datenobjekt ist kleiner als eine einstellbare Konstante $MaxDiffSize_{Client}$, die die maximale Änderung beschreibt, die bezüglich des Aufwandes der Aktualisierung der Replikate toleriert wird.

3. Der eigene Knoten, auf dem das Replikat existiert, ist nicht überlastet.

Die erste Bedingung basiert auf der Überlegung, daß die alternative Realisierung des Zugriffs auf das Datenobjekt mittels eines entfernten Zugriffs genau zwei Nachrichten benötigt und ein Replikat im Mittel weniger Nachrichten erfordern soll.

Die in den Heuristiken der beiden oben genannten Strategien verwendeten Variablen, $IntervalSize$ genannt, werden bei der Erzeugung der passiven Datenobjekte initialisiert und nach Ausführung einer bestimmten Anzahl von Zugriffsoperationen auf den initialen Wert zurückgesetzt. Dadurch wird erreicht, daß die Entscheidungen jeweils auf Informationen basieren, die den aktuellen Zugriffsmustern möglichst gut entsprechen. Zudem gibt es jeweils für beide Strategien eine untere Schranke für die Anzahl der auszuführenden Zugriffsoperationen bevor die Heuristiken ausgewertet werden dürfen. Dadurch wird gewährleistet, daß eine hinreichend repräsentative Datenmenge die Basis für die Entscheidungen darstellt. Entscheidet sich ein Client für den Wechsel von lokaler zu entfernter Nutzung, so teilt dieser dem Server bei der nächsten entfernten Nutzung die Anzahl der lokal ausgeführten Zugriffsoperationen mit. Liegt diese unterhalb einer vorgegebenen Schwelle, so beurteilt der Server die damals getroffene Entscheidung für eine Replikation nachträglich als falsch und erhöht einen Zähler für solche Fehlentscheidungen. Wenn dieser Zähler eine gewisse Schranke erreicht, vergrößert der Server den Beobachtungszeitraum ($IntervalSize$)

und erhöht somit seine Trägheit, um kurzfristige Schwankungen aus den Beobachtungen zu glätten. Andererseits werden auch die als korrekt bewerteten Entscheidungen für eine Replikation gezählt. Wird damit die Schranke erreicht, so wird die Intervallgröße wieder auf den initialen Wert zurückgesetzt, um so dynamischer auf entsprechende Zugriffsmuster reagieren zu können und die ursprüngliche Flexibilität wieder zu erlangen. Dieses adaptive und reflexive Verhalten erlaubt auch ohne vorgegebenes Wissen bezüglich der Zugriffsmuster einen angemessenen Einsatz der Heuristikauswertung.

3.2 Meßergebnisse zur dynamischen Replikation

Die einerseits zur Eichung der Heuristikparameter, andererseits zur Evaluierung der Heuristik durchgeführten Experimente untermauern die getroffenen Entscheidungen bei der Entwicklung der Strategien und rechtfertigen den Aufwand zur Bereitstellung der Möglichkeit, Realisierungen zur Laufzeit dynamisch zu ändern. So zeigt ein Experiment bei dem der Anteil der Schreiboperationen variiert wurde, daß die Realisierung mittels dynamischer Replikation erst deutliche Vorteile bringt, wenn der Anteil der Schreiboperationen unter 10 % sinkt. Zudem belegen Messungen, daß die dynamische Replikation im Vergleich zur ausschließlichen entfernten Nutzung unter Umständen auch nicht wesentlich teurer ist.

In einem weiteren Experiment wurde gezeigt, daß die Berücksichtung der durchschnittlichen Größe einer Änderung am Datenobjekt wichtig ist. Allerdings ergeben sich erst bei durchschnittlichen Änderungen im Umfang ab 400 Byte und mehr deutliche Vorteile für die dynamische Replikation, die diese Größe berücksichtigt, da ein großer Änderungsumfang einen weiter steigenden Mehraufwand erzeugt.

Ein Experiment zur Untersuchung des Nutzens der Adaptivität ergab bei einer Schreiberrate von 10 % insbesondere im Bereich der ersten 300 Zugriffe nennenswerte Vorteile, da die nicht adaptive Strategie viele unrentable Replikationen erzeugt, dies aber nicht erkennt. Die adaptive Variante hingegen beschränkt ihre Replikationsaffinität und vermeidet so unnötige Kosten. In [2] sind die dargestellten Ergebnisse der Vergleichsmessungen genauer beschrieben.

Migrationsstrategie

Für migrationsfähige Datenobjekte wurde eine Strategie entwickelt, die jeweils im Anschluß an einen erfolgten Zugriff bewertet, ob eine Migration durchgeführt werden soll. Dabei werden wieder Heuristiken zur Entscheidungsfindung herangezogen. Neben dem Optimierungsziel, die Kosten der Aufrufoperation auf dem Datenobjekt durch lokale Zugreifbarkeit möglichst zu minimieren, galt es den Rechen– und Speicherbedarf der Strategie selbst gering zu halten. Das Verfahren läßt sich in drei Phasen gliedern: eine mit geringem Aufwand verbundene Beobachtungsphase, eine Bewertungsphase und eine Migrationsphase.

In der Beobachtungsphase werden die lokal bzw. die insgesamt durchgeführten Zugriffe gezählt. Nach einer gewissen Anzahl von Zugriffen (z. B. 20) werden die Zähler zurückgesetzt. Sofern der Anteil der lokal ausgeführten Zugriffe über einer gewissen Schranke (10 %) liegt, verbleibt die Strategie in der Beobachtungsphase. Andernfalls beginnt sie die Bewertungsphase und erfaßt die aktuelle Zugriffssituation in einem Beobachtungsintervall mit einer initialen Länge von 100 Zugriffen.

Dabei werden die Zugriffe nach Knoten sortiert gezählt und zudem der Lastzustand der Knoten ermittelt. Die so gesammelten Daten werden dann zur Bestimmung eines Nutzungsschwerpunktes ausgewertet. Ergibt sich so ein lohnendes Migrationsziel, so erfolgt der Übergang zur Migrationsphase, in der dann das Datenobjekt bewegt wird und dort mit der ersten Beobachtungsphase wieder beginnt. Ergibt die Analyse des Zugriffprofiles kein sinnvolles Migrationsziel, so wird gleich wieder die erste Beobachtungsphase eingeleitet.

Um ein zu häufiges und damit zu ineffizientes Migrieren zu vermeiden, bewertet die Migrationsstrategie jede durchgeführte Migration anhand der lokal durchgeführten Zugriffsoperationen. Werden auf die Art zu viele ungünstige Migrationsentscheidungen erkannt, so reduziert die Strategie ihre Migrationsaffinität, indem zum einen das Beobachtungsintervall der Bewertungsphase vergrößert wird und zum anderen die Bewertungskriterien bei der Auswahl eines Migrationszieles verschärft werden. Steigt daraufhin die Güte der Migrationsentscheidungen (Anteil der positiv bewerteten Migrationen) wieder, so werden die erwähnten Maßnahmen rückgängig gemacht. Es handelt sich also auch hier wiederum um eine sich selbst bewertende und adaptierende Strategie. Details bezüglich der bewertenden Heuristik sind in [2] dargestellt.

3.3 Meßergebnisse zur Migration

Die zur Bewertung der Migrationsstrategie durchgeführten Experimente zeigen, daß sich bei exklusiver Nutzung recht schnell, das heißt nach einer Anzahl von etwa 150 Zugriffen, der Zeitbedarf auf etwa 50 % im Vergleich zu einer Lösung ohne Migration verringert. Im weiteren Verlauf nähert er sich asymptotisch der 10 % Grenze und liegt nach etwa 450 Zugriffen bereits bei 20 %. Bei Experimenten mit gemeinsamer Nutzung ohne Nutzungsschwerpunkt ergeben sich im Intervall bis 200 Zugriffen im Mittel 5 % höhere Kosten gegenüber einer Realisierung ohne Migration und ab dann Einsparungen in Abhängigkeit der Anzahl der zugreifenden Komponenten zwischen 5 % und 25 %.

Experimente mit gemeinsamer Nutzung und ausgeprägtem Nutzungschwerpunkt verlaufen etwa bis zum 150.ten Zugriff ähnlich. Ab dann trägt die erfolgreiche Detektion des Nutzungsschwerpunktes Früchte und senkt die Zugriffskosten auf 40 % bis 60 %. Um die Effekte der Adaptivität der Migrationsstrategie beurteilen zu können, wurden zwei Versuchsläufe durchgeführt, einer mit Adaptivität und einer ohne, wobei sich klar zeigt, daß die adaptive Variante nicht nur schneller eine Kostenreduktion erreicht, sondern auch im anfänglichen Bereich, d. h. schon bei wenigen Zugriffen, die Kosten geringer hält. Die Abbildung 2 zeigt die erzielten Laufzeiten der adaptiven Strategie (bei Phasen mit gemeinsamem und exklusivem Zugriff) bezogen auf die Laufzeiten einer Realisierung ohne Migrationsmechanismen. Dabei zeigt sich deutlich, daß die Anzahl der konkurrierenden Nutzer eines Monitor–Datenobjekts sich nur unwesentlich auf die Qualität der Migrationsstrategie auswirkt. Die in Abbildung 2 erwähnten Akteure sind die verteilt zugreifenden Nutzer der Monitor–Datenobjekte.

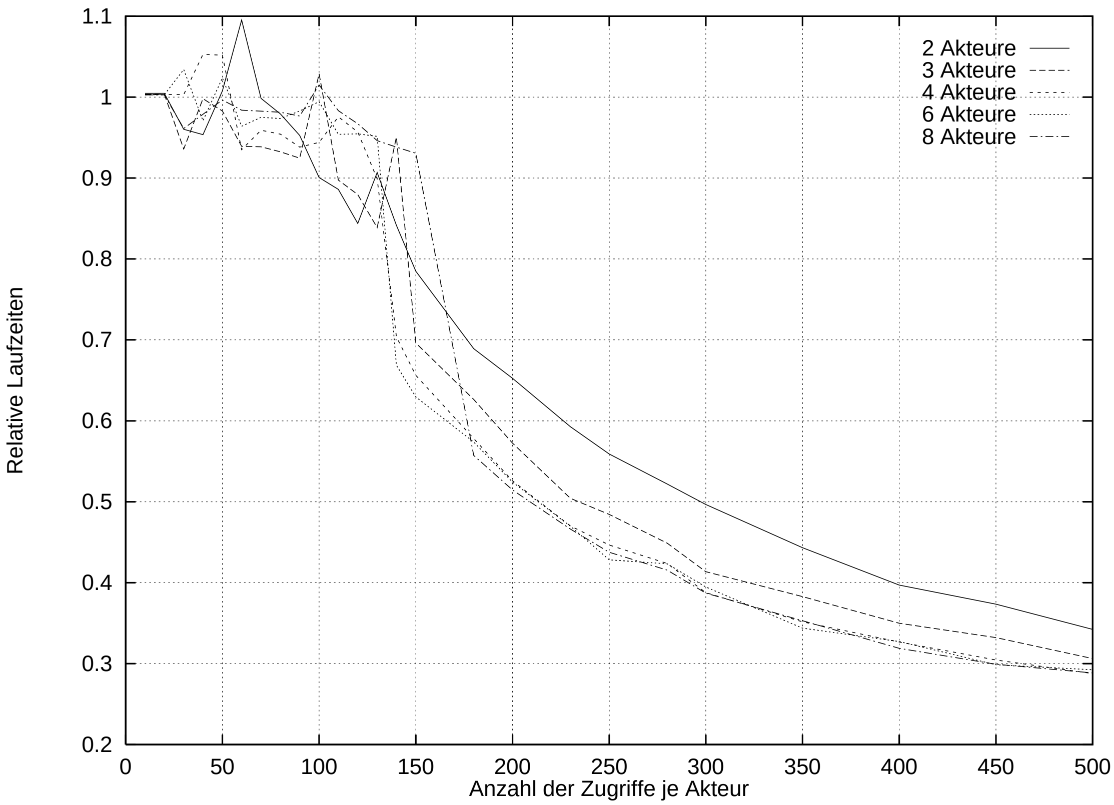

Abbildung 2: Beleg der Leistungsgewinne durch die erreichten relativen Laufzeiten bei Verwendung der adaptiven Migrationsstrategie und unterschiedlicher Nutzeranzahl

3.4 Bewertung der Strategien in AdaM

Für die Strategien der dynamischen Replikation läßt sich abschließend sagen, daß sie im allgemeinen Fall den Aufwand zur Nutzung der Datenobjekte nicht erhöhen; allerdings werden erst bei einem relativ geringem Anteil an Schreiboperationen die Vorteile der replizierten Datenobjekte bzw. des lokalen Zugriffs spürbar.

Bezüglich der Migrationsstrategie ist hervorzuheben, daß sie in der initialen Einschwingphase den Zugriff auf Monitor–Datenobjekte nur unwesentlich verteuert, aber bei ausgedehnterer Nutzung Leistungsgewinne erbringt. Dies gilt natürlich insbesondere dann, wenn ausgeprägte Nutzungsschwerpunkte vorliegen.

Den Strategien gemeinsam ist, daß sie durch eine flexible Gestaltung der Realisierung von Datenobjekten eine Leistungssteigerung erzielen und die für parallele Systeme essentielle Eigenschaft der Skalierbarkeit durch ihren verteilten Charakter gewinnen. Ferner entsteht durch die Bewertung zurückliegender Entscheidungen und durch die eigene Adaption ein rückbezügliches Verhalten, das den Strategien die nötige

Anpassungsfähigkeit einbringt, welche für Problemlösungen ohne a priori Wissen gefordert ist.

Weiterführend wären untersuchungswerte Verbesserungsmöglichkeiten z. B. die Fragestellung, ob man die schrittweise vorgenommenen Anpassungen, beispielsweise der Intervallgrößen, nicht auch schrittweise wieder zurücknehmen sollte, statt sie wie bisher auf den initialen Wert zurückzusetzen.

Zusammenfassend ist hinsichtlich der Integration der Strategien in ein verteiltes Ressourcenmanagement zu sagen, daß erste Erkenntnisse und Experimente mit vielversprechenden Resultaten im Bereich der Anpaßbarkeit und der Nutzung eines Realisierungsspektrums für Komponenten gemacht wurden. Allerdings darf nicht verschwiegen werden, daß in der vorliegenden Arbeit nicht alle INSEL–Konzepte umgesetzt wurden. So wird beispielsweise zwar die Auslastungssituation der Hardware–Knoten im Entwurf bedacht, jedoch nicht bei der Implementierung des Experimentalsystems umgesetzt.

4 Ausblick

Für die zukünftigen Entwicklungen zeichnet sich klar die Zielsetzung ab, die untersuchten Verfahren zu fusionieren und die gewonnenen Ergebnisse dadurch einfließen zu lassen. Es sollten einerseits zur Lastplazierung von Aktivitätsträgern, Migrationsmechanismen mit reflexivem und adaptivem Verhalten hinzugenommen werden. Andererseits sollte die Last der Hardware–Knoten auch in die flexiblen Realisierungsentscheidungen von Datenobjekten einfließen. Darüber hinaus wurde in AdaM in Abhängigkeit des zu realisierenden Datenobjektes nur jeweils eine Alternative (dynamische Replikation bzw. Migration) gegenüber der entfernten Nutzung durch RPC–Mechanismen zur Wahl gestellt. Eine Untersuchung, welche Vorteile eine Auswahl der Realisierung aus mehreren Möglichkeiten für ein Datenobjekt eröffnet, wurde nicht durchgeführt. Es erscheint jedoch sinnvoll für die Realisierung eines Datenobjektes ein Spektrum an Realisierungsfreiheiten und Möglichkeiten einzuräumen. So läßt sich zum Beispiel das folgende typische Szenario am besten durch dynamische Replikation und mit anschließender Migrationsphase realisieren. Zu Beginn werden Daten als Parameter einer parallelen Berechnung aus einer Datenstruktur von den verteilten Aktivitätsträgern gelesen, dazu bietet sich die Replikation an, sofern die Phase des gleichzeitigen Lesens hinreichend lang bzw. rentabel ist. Wenn dann die Berechnungen nicht gleichzeitig enden, kann es sich als sinnvoll erweisen, die Replikate aufzugeben, das Datenobjekt migrationsfähig umzusetzen und durch Migration zu den entsprechenden Stellen die Ergebnisse mit wenig Aufwand zu sammeln.

In beiden Experimentalsystemen wurde das Potential der Information aus der Abhängigkeitsstruktur, das implizit durch die Konstruktionskonzepte verfügbar ist, nicht ausgenutzt, um Realisierungs– oder Plazierungsentscheidungen zu beeinflussen. Ein zukünftiges, verteiltes, systemintegriertes Ressourcenmanagement darf solche wertvollen Möglichkeiten nicht brach liegen lassen und sollte eine Verschränkung und engere Kooperation der einzelnen zu verwaltenden Ressourcen beinhalten. Zusätzlich gilt es dann auch die wechselseitigen Auswirkungen zwischen den Managementmaßnahmen zu betrachten, um Reibungsverluste zu vermindern und Synergieeffekte zu nutzen. So werden beispielsweise in [6, 7] teilweise solche Wechselwirkungen zwischen Cache–Speichermanagement und dem Scheduling beschrieben sowie Verfahren zu deren Berücksichtigung und Nutzung der Wechselwirkungen. Diese Verknüpfung soll in der zukünftigen Managementarchitektur auf einen noch weitergefaßten Ressourcenbegriff ausdehnt werden, der dann beispielweise auch Kommunikationskanäle, Dienste anbietende Komponenten einschließt.

Zusätzlich werden die Managementstratgien der zukünftigen Entwicklung ein reflektiv adaptives Verhalten zeigen, das heißt, sie werden ihre eigenen Entscheidungen selbst bewerten und sich gemäß dieser Einschätzung selbständig anpassen bzw. verändern. Dieser Ansatz fand sich bereits geringfügig ausgeprägt in AdaM und auch Erfahrungen [8] im Bereich von Parallelrechnern belegen die erzielbaren Vorteile hinsichtlich einer Erhöhung der Auslastung bzw. der Einsparung von Rechenzeit.

Weitere Erkenntnisse, die sich auf andere Arbeiten in diesem Teilprojekt auswirken, sind die erfahrenen Begrenzungen bei der Realisierung von INSEL–Systemen durch die Verwendung von Standardwerkzeugen, wie dem C–Compiler oder von Thread–Paketen zur Realisierung der Aktivitätsträger. Durch den C–Compiler läßt sich die Strukturierungsinformation der Sprachkonzepte nur begrenzt in den erzeugten Objekten umsetzen und nutzen. Analog fehlt die für anwendungsangepaßte Aktivitäten nötige Flexibilität durch die Realisierung von Aktivitätsträgern mit Hilfe von starren Thread–Paketen. Mit dem verfolgten sprachbasierten Gesamtsystemansatz entwickeln wir einen verteilten Stellenkern [9] sowie einen Compiler [10], der dynamisches Laden und Binden unterstützt, so daß zur Laufzeit die Realisierungen auf einem sehr niedrigen Niveau noch angepaßt werden können und die zugrundeliegenden Basisdienste des Kerns auf die Anforderungen des Ressourcenmanagements abgestimmt sind.

Literatur

[1] *Ralph Radermacher*. „Eine Ausführungsumgebung mit integrierter Lastverteilung für verteilte und parallele Systeme“. Dissertation, Technische Universität München (1995).

[2] *Hans Michael Windisch*. „Speicherverwaltung für konzeptionell strukturierte verteilte Systeme“. Dissertation, Technische Universität München (1996).

[3] *P. P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer und H.-M. Windisch*. Sprachkonzepte zur Konstruktion verteilter Systeme. Bericht, TU München (März 1996). SFB–Bericht 342/09/96 A TUM–I9618.

[4] *R. Radermacher und F. Weimer*. INSEL Syntax–Bericht. Bericht, TU München (März 1996). SFB–Bericht 342/08/96 A TUM–I9617.

[5] *J.A. Stankovic und I.S. Sidhu*. An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups. In „Proceedings 4th International Conference on Distributed Systems“, Seiten 49–59 (1984).

[6] *F. Bellosa*. Locality-Information-Based Scheduling in Shared-Memory Multiprocessors. In *L. Rudolph D. Feitelson* (Herausgeber), „IPPS'96 Workshop of Job Scheduling Strategies for Parallel Processing“, Band 1162 aus „LNCS“, Seiten 271–289, Honolulu, Hawaii (April 1996).

[7] *F. Bellosa*. The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing* **37**(1), 113–121 (August 1996).

[8] *Niels Reimer, Stefan U. Hänßgen und Walter F. Tichy*. Dynamically Adapting the Degree of Parallelism with Reflexive Programs. In „Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR)“ (1996).

[9] *Christian Czech*. Architektur und Konzept des Dycos–Kerns. Bericht, TU München (April 1997). SFB–Bericht 342/08/97 A TUM–I97XX.

[10] *Markus Pizka*. Design and Implementation of the GNU INSEL–Compiler (gic). Bericht, TU München (April 1997). SFB–Bericht 342/09/97 A TUM–I9713.

[11] *Th. Schnekenburger und G. Stellner (Eds)*. „Dynamic Load Distribution for Parallel Applications“. Institut für Informatik, Technische Universität München (to appear).

[12] *N. Reimer und M. Pizka*. „Dynamic Load Distribution for Parallel Applications“, Kapitel Load Distribution Strategies of the Distributed Thread Kernel DTK. (to appear).